Service component - A mechanism for web service composition reuse and specialization

Orriëns, B.; Yang, J.; Papazoglou, M.

Published in:
Proceedings of the 7th World Conference on Integrated Design and Process Technology

Publication date:
2003

Integrated Design and Process Technology, **IDPT-2003**
Printed in the United States of America, June, 2003

SERVICE COMPONENT: A MECHANISM FOR WEB SERVICE COMPOSITION REUSE AND SPECIALIZATION

Bart Orriëns, Jian Yang and Mike P. Papazoglou
Department of Information Management
Tilburg University
PO Box 90153, 5000 LIE, Tilburg, Netherlands

ABSTRACT

Web services are becoming the dominant paradigm for distributed computing and electronic business. This has raised the opportunity for service providers and application developers to create value added services by combining web services. Several web service composition solutions have been proposed, e.g. WSFL and BPEL4WS. However, none of the existing solutions addresses the issue of service composition reuse and specialization, i.e. how applications can be built upon existing simple or composite services by reuse, restriction or extension. In this paper we propose the concept of Service Component that packages together elementary or complex services and provides composition logic and semantics. Within the framework of Service Components we examine different aspects of composition reuse and specialization. A specification for web component reuse and specialization is provided, and possible solutions to support this specification are presented. To demonstrate our framework we provide an overview of ServiceCom, the tool that implements the Service Component concept, supporting reusable web service composition specification, combination, and execution.

INTRODUCTION

The Web has become the means for organizations to deliver goods and services and for customers to search and retrieve services that match their needs. Web services are self-contained, web-enabled applications capable not only of performing business activities on their own, but also possessing the ability to engage other web services in order to complete higher-order business transactions. Examples of such services include catalogue browsing, ordering products, making payments and so on. The platform neutral nature of the web services creates the opportunity for building composite services by using existing elementary or complex services possibly offered by different enterprises. For example, a travel plan service can be developed by combining elementary services such as hotel reservation, ticket booking, car rental, sightseeing package, etc., based on their WSDL descriptions. By composite service, we mean a service that uses other services. The services that are used in the context of a composite service are called its constituent services.

Web service design and composition is a distributed programming activity. There are cases where people want to reuse the design and implementation of the web services only by extension or restriction without developing them from scratch. Therefore the service based distributed application design process is very similar to the original vision of component based development, not purely about development, but about managing the assembly of collaborative assets to provide a highly adaptive application solution. This requires software engineering principles and technology support for service reuse, specialization, and extension such as those used, for example, in component based software development. However it is not possible at the moment to define and implement a web service composition once, and use it in similar designs with some variations. Although web services provide the possibility for offering new services by specialization and extension their WSDL specification, to this date there is little research initiative in this area, let alone mechanism and specification supported.

The basic idea behind composition reuse is simple: develop web service composition specifications, then reuse and specialize them in similar composition conditions. In the object-oriented world, the basic elements for reuse are object types and classes, and reuse and specialization are achieved by inheritance and sub-typing. While for the component systems in which the reusable objects are components, reuse and specialization can be realized by various mechanisms such as inheritance, extension, configuration, parameters, etc.

In this paper we introduce the concept of ***service component*** to facilitate this very idea of web service component reuse, specialization and extension, and discuss why the inheritance concepts developed by object-oriented programming language research cannot be applied directly to service component inheritance but with

modification. ***Service components*** are a packaging mechanism for developing web-based distributed applications in terms of combining existing (published) web services. Service components have a recursive nature in that they can be composed of published web services while in turn they are also considered to be themselves web services (albeit complex in nature). Once a service component is defined, it can be reused, specialized, and extended. In this paper we briefly describe the service component framework for web service composition, analyze the characteristics of service component and its aspects for reuse and specialization. Following this we discuss the specification requirements for service component reuse and specialization and possible system support. Then, to demonstrate our work we provide an overview of **ServiceCom**, the tool we developed that implements our framework for reusable web service composition specification, combination, and execution.

SERVICE COMPONENT FOR WEB SERVICE COMPOSITION

As a starting point for the discussion of composition reuse and specialization, we need to identify several key elements in composition specification, which are commonly expressed in the currently proposed standard such as Business Process Execution Language for Web Services (BPEL for short) [1] and business process modeling language (e.g., BPML [2], XPDL [3]). We then explain how these key elements are supported and packaged in service components. Service components then can be used as building blocks for developing web applications based on composing service functionality. Consequently, the process of developing web service composition becomes a matter of reusing, specializing, and extending the available service components. This enables a great deal of reusability of service compositions. To understand how this happens we need to look at the inner structure and inner workings of a service component. In the following sub-sections, we will first introduce the elements of the service components; then we present the basic constructs for service component creation.

Service component elements

In most proposed web service composition and business process modeling specifications, the basic elements are activities, data and control. In BPML the control has also been modeled as "activity", within which a set of activities are included. In this block-structured (as opposed to activity-transition like WSFL [4]) approach, each block can itself be considered an activity. For example, if then else is modeled as a block activity, which contains a set of alternative activities based on the conditions. Blocks can then be nested to arbitrary levels. The fact that the control is expressed as activities allows the management of the context of these activities at a very fine level. This is also a good foundation for transactional semantics and provides a substrate for monitoring the state of the process. Most importantly, the use of block structures makes reuse easier. Naturally the proposed service component is block-structured. However, it extends the existing block-structure in the following aspects:

- Service component packs activity blocks together and define an interface for them, which can be published as a web service.
- A "block" in the service component has semantically richer context then normal control block (e.g., sequential, if then, etc). It specifies how a composition is constructed in terms of conditions, order, and alternatives.
- A "block" in the service component is one of the important aspects for reuse and specialization.

Figure 1 depicts the ingredients of a service component. It illustrates that a ***service component*** presents a single public interface to the outside world in terms of a uniform representation of the exported functionality of its constituent services and the input/output messages, It also internally specifies its composition logic in terms of composition type and message dependency constructs. ***Composition logic*** refers to the way a composite service is constructed in terms of its constituent services.

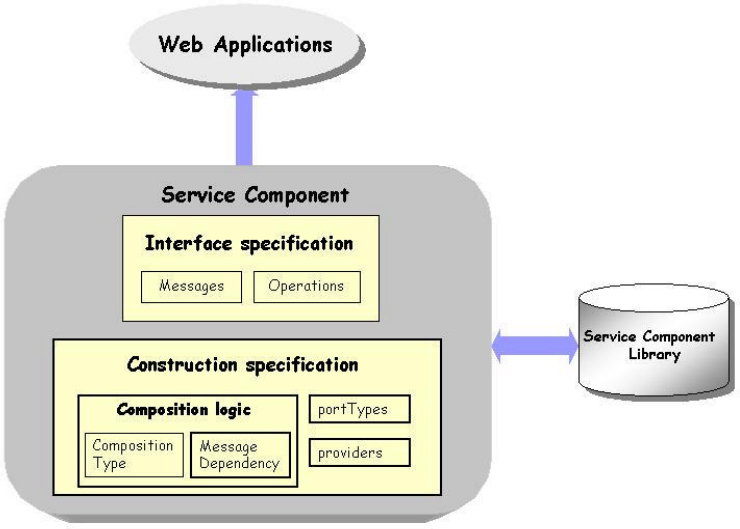

Fig. 1: Service Component Ingredients

A block of web service operations is composed according to the composition logic of the block in a service component. Here, we assume that all publicly

available services are described in WSDL [5]. Composite logic comprises the following two fundaments:

- ***Composition type***: this construct signifies the nature of the composition that can take the combination of the following three forms:
  - *Order*: this construct indicates whether the constituent services in a composition are executed in a serial or parallel manner.
  - *Alternative service execution*: which indicates whether alternative services can be invoked in a given service composition. Alternative services can be tried either in a sequential or in a parallel manner.
  - *Conditioned execution*: which represents if then and while do constructs.
- ***Message dependency***: this construct signifies whether there is a message dependency among the constituent services and the composite service. We distinguish between three types of message dependency handling necessary for a composition:
  - *Message synthesis*: this construct combines the output messages of constituent services to form the output message of the composite service.
  - *Message decomposition*: this construct decomposes the input message of the composite service to generate the input messages of its constituent services.
  - *Message mapping*: this construct specifies the mappings among the inputs and outputs of the constituent services. For example, the output message of one constituent service could be the input message of another service.

What remains to be examined is the core constructs for expressing composition logic, how they are defined and how they can be reused and specialized. These issues are discussed in detail in the following section.

SERVICE COMPONENT ELEMENTS

In this section, we first present the basic constructs for expressing composition logic; then we illustrate how service components are defined in terms of these constructs.

Basic constructs for service composition

The following constructs have been identified to serve as the basis for compositions [6]:

1. ***Sequential service composition (sequ).*** In this case, the constituent services are invoked successively. The execution of a constituent service depends on its preceding service, i.e., one cannot begin unless its preceding service has committed. For example, when a composite travel plan service - composed of an air ticket reservation service, a hotel booking service, and a car rental service - makes a travel plan for a customer, the execution order should be hotel booking, air ticket reservation, and car rental. The invocation of the hotel booking service is dependent on a successful execution of the air ticket reservation, because without a successful air ticket reservation, hotel booking cannot go ahead.
2. ***Sequential alternative composition (seqAlt).*** This situation indicates that alternative services could be part of the composition and these are ordered on the basis of some criterion (e.g., cost, time, etc). These will be attempted in succession until one service succeeds.
3. ***Parallel service composition***. In this case, all the component services may execute independently. Here, two types of scenarios may prevail:
   a. *Parallel with result synchronization* (*paraWithSyn*). This situation arises when the constituent services can run concurrently, however, the results of their execution need to be combined. For example, purchasing a PC may involve sending inquiries to different companies, which manufacture its parts. These inquires may run in parallel, however, they all need to execute to completion in order to obtain the total configuration and price.
   b. *Parallel alternative composition* (*paraAlt*). In this situation alternative services are pursued in parallel until one service is chosen. As soon as one service succeeds the remainder are discarded.
4. ***Condition***. In this case, condition is used to decide which execution path to take.
5. ***While do***. This is a conventional iteration construct with testing condition.

The various composition types may result in different message dependencies and therefore require different message handling constructs. Table 1 summarizes the message dependency handling constructs required for different types of service composition. A cross "X" in the table indicates the possible message dependency required to specify for a particular construct. For example, in case

of seqAlt type of composition, message mapping may be needed. The basic composition types together with message dependency handling constructs provide a sound basis for forming the composition logic in a service component.

	Message Synthesis	Message Decomposition	Message Mapping
Sequ	X	X	X
SequAlt			X
ParaWithSyn	X	X	X
ParaAlt			X
Condition			X
WhileDo			X

Table 1: Message handling in composition types

Service component class definition

To be able to reuse and specialize service components, we specify service components as classes. In Figure 2 below we define a service component class for a service travelPlan. We assume that a travel plan is a composite service which combines three services hotelBooking, ticketReservation, and sightSeeing which are provided by hotelBookingProvider, ticketReservationProvider, and sightSeeingProvider respectively, and presumably have their own input and output messages and operations defined in their service component classes. All the message types are defined in their XML Schemas. A service component can be specified in two isomorphic forms: A class definition and an XML/WSDL specification that corresponds and conforms to the class definition of a service component and is used to implement the class. The class definition of a web service is used for discovery, reuse, extension, specialization, and versioning purposes, whereas its corresponding XML/WSDL form is used for construction purposes, message exchange, service invocation and communication across the network. A complete XML/WSDL specification of this example can be found in [6]. In the next subsection we will analyze various aspects in the service component class for reuse and specialization.

serviceComponentClass TravelPlan {

Definitions

d1: TripOrderMessage tripOrderMsg

d2: TripResultMessage tripResDetails

o1: travelPlanning (in tripOrderMsg, out tripResDetails)

Construction

c1: sequ(HotelBooking.booking,TicketReservation.ticketRes

c2: ParaWithSyn(c1, SightSeeing.sightseeing)

Providers

p1: HotelBooking HotelBookingProvider

p2: TicketReservation TicketReservationProvider

p3: SightSeeing SightSeeingProvider

MessageHandling

m1: messageDecomposition(TravelPlan.tripOrderMsg,Hotel-Booking.hotelBookingMsg, TicketReservation.ticketResMsg)

m2: messageSynthesis(HotelBooking.hotelBookingDetails, TicketReservation.e-ticket, SightSeeing.sightSeeingDetails)

}

Fig. 2: A Service Component Class Definition

REUSE ASPECTS IN SERVICE COMPONENTS

There are five reuse and specialization aspects based on the service component class definition in Figure 2: **Definition**, **Construction**, **Providers**, **MessageHandling,** and the service component class itself. In the following we will discuss the possible ways of reusing and specializing a service component class, and the requirements in the specification.

To illustrate the first four aspects let us look at the following example: Suppose a specific travel planning service includes a fourth service RestaurantBooking that can be done in parallel with others. Instead of defining a new composite service from scratch, we can extend the service component class in Figure 2 by introducing a subclass AddTravelPlan, in which the construction c3 is added to the **Construction** section and a provider for the new service in the **Providers** section. We also overwrite the content of the **MessageHandling** section to include the input and output messages results of the RestaurantBooking service in the decomposition and synthesizing process. The extended class definition is depicted in Figure 3.

serviceComponentClass AddTravelPlan

SubclassOf TravelPlan {

Construction

c3: ParaWithSyn(c2, RestaurantBooking.resBooking)

Providers

p4: RestaurantBooking RestaurantBookingProvider

MessageHandling

m1: messageDecomposition(TravelPlan.tripOrderMsg,Hotel-Booking.hotelBookingMsg, TicketReservation.ticketResMsg, RestaurantBooking.restaurantBookingMsg)

m2: messageSynthesis(HotelBooking.hotelBookingDetails,
TicketReservation.e-ticket, SightSeeing.sightSeeingDetails,
RestaurantBooking.restaurantDetails)
}

Fig. 3: A Service Component Class Definition

Service Component Reuse and Specialization

Since a service component is specified as a class, it can be reused whenever a service is required. If any part of the service component is used, e.g., its messages and operations, then they have to be prefixed with the name of the service component. As illustrated above, overriding and extending any service component aspects is a specialization of a service component. The keyword SubclassOf means that the defining service component class inherits all the aspects from the super service component class, except the points that are redefined and modified in the subclass. Any new names introduced in any aspects for the subclass service component, which are not defined in the superclass, are treated as extension aspects to the superclass. All the names appearing in the subclass, which are also used in the superclass, are treated as overwriting of the same aspect definitions in the superclass.

SERVICE COMPONENT CLASS LIBRARY

Service component classes are used as a mechanism for packaging, reusing, specializing, extending and versioning web services (or service components). In this section, we will discuss how service components are created and how they are used in web service composition based application development.

Service component creation

There are generally two approaches for creating service components:

- ***Converting from a WSDL/XML specification***: in this case, a published WSDL web service or a XML based service composition (e.g., in BPEL) is converted into an equivalent object-oriented notion (class) as illustrated in the previous section. Any kind of web service (composite or not) can be represented and stored as a service component class provided by an organization and can be used in the development of distributed applications.
- ***Defining a service component class from scratch***, in which case there are two scenario:
  - *Defining a concrete service component class*, i.e. all the aspect elements in the service component are provided: Definitions, Construction, MessageHandling, possibly service providers as well. The example in Figure 2 is a concrete class.
  - *Defining an abstract service component class*, i.e., only Definitions aspects are specified in the abstract class. Concrete service component classes can be used to implement the abstract classes. For example, we can define an abstract service component class as followed:

serviceComponentClass aTravelPlan Abstract {

Definitions

TripOrderMessage tripOrderMsg

TripResultMessage tripResDetails

operation travelPlanning (in tripOrderMsg, out tripResDetails)

}

There are two ways to realize an abstract class by concrete classes: (1) defining concrete classes as the subclass of the abstract classes. The example in Figure 2 can be defined as a subclass of the above abstract class. In this case, the concrete subclasses not only realize the abstract class, but also can extend the definition, for instance, adding more operations.

(2) Defining concrete classes by implementing the abstract class. Similar to Java, in this case, we can have several service component classes implement the same abstract class. We can also have a single service component class implements several abstract service component class. For example, we can define the following two concrete service component classes which implements the abstract class aTravelPlan:

serviceComponentClass DomensticTravelPlan implements aTravelPlan {

}

serviceComponentClass InternationalTravelPlan implements aTravelPlan {

}

In this case, the two concrete classes have to have the same Definitions as the abstract class. Additional binding rules need to be specified so that the system knows which implementation to pick up at run time when the abstract class is used.

Service component class library and repository

The service component class library is a collection of general purpose and specialized classes implementing the primitives and constructs discussed in the previous section. Classes in the service component library act as

abstract data types i.e., they cannot be instantiated. They provide basic constructs and functionality that can be used to create service component classes. The service component library classes can be categorized as followed:

- ***Service component creation classes***: these classes are used for creating service component classes out of WSDL/XML specifications. Service component classes for registered web services defined in WSDL will be created by using the construction classes, i.e., the Definitions, Construction, MessageHandling, and Providers will be generated for the service component classes.
- ***Service component construction classes***: these classes provide the semantics and functions to implement the composition constructs discussed in the previous section.
  These classes are: *Sequ*, *SeqAlt, ParaWithSyn, ParaAlt, IfElse*, and *WhileDo*. The construction classes are used for building new service components.

The service component class repository has a collection of concrete service component classes, service component application classes. The example in Figure 2 is a service component application class. All the classes stored in the repository also have WSDL web service description published in UDDI, for instance. Therefore they can be searched and discovered. Developing web service oriented applications becomes a job of reusing and specializing service component application classes. The various types of service component classes and their relationships are summarized in Figure 4.

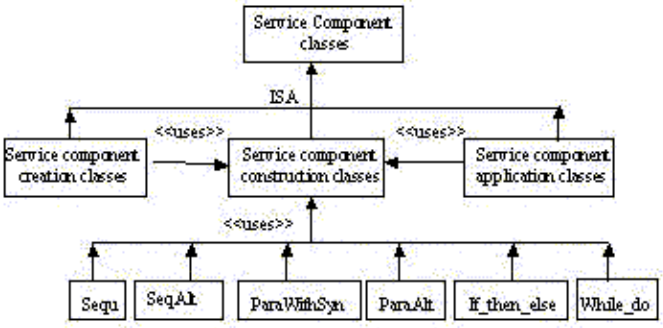

Fig. 4: Service Component Classes

SERVICE COMPONENT IMPLEMENTATION

To test and demonstrate reuse and specialization of web service compositions through service components we developed the **ServiceCom** tool. **ServiceCom** is a Java based implementation of the web service composition phase model [7], as depicted in Figure 5. This model provides a high-level description of the web service composition process. ServiceCom utilizes the service component mechanism as its underlying mechanism to support this model. For this purpose a **Service Composition Specification Language (SCSL)** has been developed (see [7] for the syntax). In this language a service component presents a single *public interface* to the outside world in terms of a uniform representation of the features with exported functionality of its constituent services. Internally it specifies a set of web services, which are choreographed in a particular manner. *Activity* constructs represent these constituent services through the use of name and description characteristics. To bind activities to particular web service providers *binding* constructs can be attached to activity constructs. These bindings utilize information defined in WSDL. To be exact, they refer to a single operation on a port of a service in the WSDL interface of the provider. Alternatively, bindings may provide search criteria to enable the locating of appropriate providers during runtime (for example if a suitable provider is not known to the user). *Condition* constructs may be used in case of conditional compositions to govern the control flow within the service component. To express different forms of activity choreographies SCSL utilizes the *composition type* construct. Supported types are based on the basic service constructs defined in section 3.1 and include *If*, *IfElse*, *ParaWithSync*, *ParaAlt*, *SeqAlt*, *SeqNoInt*, *SeqWithInt* and *WhileDo*.

In the following we first describe the implementation of the web service composition phase model in ServiceCom. We then discuss the ServiceCom support for reuse, extension and specialization.

Web service composition phase model

As can be seen in Figure 5 the web service composition phase model consists of four phases: the *description*, *planning*, *building* and *invocation* phase. We briefly discuss each phase with regard to its implementation in ServiceCom.

Description phase

In the *Description* phase users can specify web service compositions. For this purpose ServiceCom offers a **composition designer**, which provides a visual-oriented development environment. *Activity*, *binding* and *condition* constructs may be dragged and dropped on the designer window. Characteristics may be set and edited through the use of popup dialogs, which can be opened by double-clicking a composition construct. Alternatively, constructs may be inserted, edited and removed through the use of menus. A screenshot of the composition designer is provided in Figure 6 in Appendix A.

Planning phase

The *Planning* phase is concerned with binding the activities in the service composition to concrete services. For this purpose ServiceCom offers a **web service provider library** facility. This library facility contains locally known web service providers, which can aid the user in case he does not know an appropriate provider for an activity in a composition. (i.e. the WSDL interfaces are stored on the local system). Web service providers may be added and removed from the library. Also, a search on the Internet can be performed to locate suitable providers, based on user-specified search criteria. Any located providers may then be added to the library, making them locally available. Furthermore, provider functionality may be tested through a quick invocation. As such, the web service provider library is of great help in the planning phase of the web service process model. For a screenshot of the library see Figure 7 in Appendix A.

In case a user does not want to search and select a service provider manually ServiceCom is capable of selecting appropriate providers automatically during runtime. For this purpose the tool utilizes the search criteria specified in the binding construct(s).

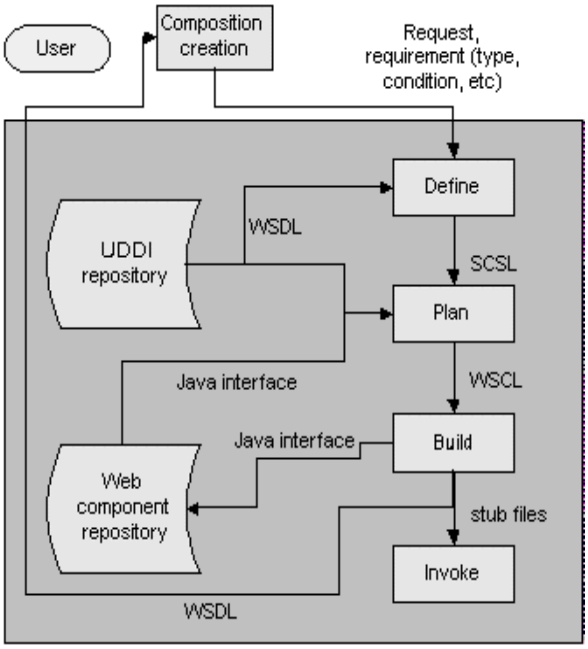

Fig. 5: The Web Service Composition Phase Model

Building phase

The *Building* phase is of essence in the web service composition phase model. It crosses the gap between composition specification and actual invocation. In this phase the web service composition specification is used as input to generate the set of Java source and class files required for the invocation of the composition. This set of files comprise of the following:

- Firstly, for each activity in the composition a set of Java source and class files is created for invocation. These files are derived from the WSDL definition of the service provider, specified in the binding construct of the activity. For each service in the definition a class is defined, as well as for the ports of these services. Operations are represented as Java methods, whereas faults are expressed as exceptions. Additionally, files are generated that translate operation invocations from and to SOAP messages to enable XML-based invocation.

- Secondly, a set of Java source and class files is created for the web service composition. These files enable the invocation of the composition in the specified manner. The exact types of files that are generated depend on the type of the composition. For example, if a composition is of the type *if*, then its condition constructs are mapped to source and class files. In case of *parallel* compositions source and class files are generated that enable the invocation of an activity in a separate thread of execution.

- Thirdly, a WSDL definition is generated for the web service composition. This definition functions as the public interface of the composition.

Invocation phase

The *Invocation* phase deals with the execution of a web service composition and its corresponding result handling. In most cases users will likely want to incorporate the composition's functionality in custom made programs. In these situations the development of nice, user-friendly interfaces are left to the designers of these programs. However, some users may simply want to invoke the service once and are not interested at all in software program development. These users may employ the standard interfaces for the web service composition, which can be generated prior to invocation. Although these interfaces are basic, they provide a means to the user to enter the required inputs and display the received outputs. We expect that a standard for building web service user interfaces based on WSDL definitions will be developed in the near future. If this is the case, then ServiceCom will be adapted to include support for this standard.

Support for reuse, extension and specialization

Reuse, extension and specialization of constructs in the *Description* phase in ServiceCom is based upon the use of unique names and namespaces. This combination ensures that a reference to a construct may only refer to a

single construct, since a construct name must be unique within a namespace and each composition has a unique namespace. For the reuse of constructs simple references can be used. Extension and specialization is supported through the utilization of base types. When specifying a base type for a construct the characteristics of this base construct are inherited by the sub-construct. Additional information in the sub-construct not present in the base construct extends the base construct. Similarly, in case of information that is present in both the sub and base construct the characteristics of the base construct are overridden. In ServiceCom both references and base types can be made through the selection of composition constructs from the **web service composition library**. (See Figure 8 in Appendix A for a screenshot)

In section 3 we distinguished five aspects of reuse, extension and specialization for service components. Table 2 shows an overview of the ServiceCom support for each of these types through the use of, among others, the above-described mechanism of references and base types.

Reuse aspect	Supported
Definition	Partly
Construction	Yes
Provider	Yes
Message handling	No
Service component	Yes

Table 2: Support for reuse, extension and specialization

As table 2 illustrates ServiceCom supports four of the five defined reuse aspect. With regard to the **Definition** aspect operations may be reused, extended and specialized through respectively the reuse, extension and specialization of the WSDL interface that is generated for the web service composition. Since message constructs are not used in ServiceCom, there is no support for their reuse, extension or specialization.

Reuse, extensions and specialization of basic constructions in the **Construction** aspect is supported. Compositions may not contain other compositions directly in ServiceCom, but references to such compositions can be made through the use of *activity* constructs. As such, reuse, extension and specialization of basic constructions becomes a matter of reusing, extending and specializing activity constructs. As mentioned earlier *binding* constructs are used in ServiceCom to bind activities to a certain web service provider. These *binding* constructs may be reused, extended and specialized in a similar fashion as *activity* constructs. As such, reuse, extension and specialization in the **Provider** aspect is accommodated. Finally, service component as a whole or their constructs can be reused, extended or specialized.

Besides the reuse, extension and specialization issues outlined in section 3 ServiceCom enables a further form of reuse in its implementation of the building phase. As stated in the discussion, at this phase it pertains to the generating of Java source and class files based on the specified web service composition to enable its invocation. These files are referred to as service component application classes in section 5 and they are reusable in the building phase. These files can be reused in composition building. Source and class files for a composition will be generated only if they are not already in the library. For example, suppose a service is involved in multiple compositions, instead of generating new files for each composition only one set files will be generated and it will be reused in all other compositions this service is involved in.

Furthermore, ServiceCom enables the reuse of application classes in two other ways: (1) Generated files can be used in the development of other applications, (2) the mentioned functionality may be extended and specialized in an object-oriented fashion through the use of Java overriding and inheritance mechanisms. However, these types of reuse are in our opinion best facilitated in professional programming environments and are thus considered to be outside the scope of ServiceCom.

Finally, reuse of the generated public interface of the composition is possible. Firstly, the WSDL definition may be included in the web service provider library, making it available for use in other compositions. Secondly, the interface file can be published in for example an UDDI registry to make the composite service's functionality known to the outside world. For the latter purpose ServiceCom offers a deploy mechanism. This mechanism enables the publication of the new, composite web service in standard web service containers, such as Tomcat and J2EE.

RELATED WORK

Sub-typing and inheritance is widely supported in object-oriented systems. However, in object-oriented system, the concepts involved in sub-typing and inheritance are classes, attributes, and methods. As we analyzed in this paper there are more aspects in a service component to be considered for reuse and specialization. BPML, XPDL, and BPEL4WS are three representing standards for XML-based process definition languages. BPML is a block-structured programming language. It supports complex activities, which refer to activity sets. The control and routing are defined within the blocks. Since a complex activity can appear in an activity set, BPML supports nesting and recursive process definition.

XPDL on the other hand is a graph-structured language. Because it only allows process definitions on the top level, there is no support for nested processes in XPDL. Activity attributes can be used in XPDL to specify the resource(s) required to perform the activity. This expression will be evaluated at execution time to determine the resource required. This feature is missing in BPML and BPEL. BPEL4WS is a block-structure programming language, which allows recursive block but does not support nested process definition. BPEL4WS is designed specifically for web service orchestration. However, none of the proposing standards has addressed the issue of service composition reuse and specialization. Some similar work has been done in the area of work class inheritance. [8] presented a framework to analyze the requirements for supporting work class inheritance. Different perspectives of inheritance were discussed and a work class definition language is proposed. In [9] the authors presented a work class specification. It consists of a set of object classes as well as a set of rules. Subclass and inheritance is supported, at least partially. [10] described a system called TOWE. It implements a set of classes, which provide the basic mechanism for work execution. Normal work classes can then be developed through inheritance of the functionality of these basic classes. Although all these work are relevant, the basic difference between the work done in the work and what we are proposing in this paper is that all the aspects in the service component class are the reuse point: they can be referred to inside or outside the class definition, consequently they are the points which can be specialized and override because aspect elements are uniquely labeled.

SUMMARY

In this paper, we present a framework for analyzing the issues around web service composition reuse and specialization. The concept ***service component*** is proposed, which is used as a packaging mechanism for composing web services. Once service component classes are defined, they can be reuse and specialized by adding and overriding the definitions of the service component aspects. Subsequently we discuss **ServiceCom**, an implementation of the service component mechanism, which enables composition development, reuse and specialization, as such covering the web service composition phase model.

REFERENCES

[1] F. Curbera, Y. Goland, J. Klein, F. Leymann, D. Roller, S. Thatte, S. Weerawarana; "Business Process Execution Language for Web Services", July 31, 2002, http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/

[2] Business Process Modelling Initiative; "Business Process Modeling Language", http://www.bpmi.org

[3] Workflow Management Coalition; “XML Process Definition Language”, May 22, 2001, http://www.wfmc.org/standards/docs/ xpdl 010522..pdf

[4] F. Leymann; "Web Service Flow Language", 2001, http://www.ibm.com/software/solutions/webservices/pdf/WSFL.pdf

[5] E. Christensen, F. Curbera, G. Meredith, S. Weerawarana; “ Web Service Description Language”, 15 March 2001, http://www.w3.org/TR/wsdl

[6] J. Yang, M.P. Papazoglou, W.J. v/d Heuvel; "Tackling the Challenges of Service Composition", ICDE-RIDE workshop on Engineering E-Commerce/E-Business, San Jose, USA, 2002

[7] J. Yang, M.P. Papazoglou; "Web Components: A Substrate for Web Service Reuse and Composition", Proceedings of 14th Conference on Advanced Information Systems Engineering (CAiSE02), Toronto, May 2002.

[8] C. Bussler; "Work Class Inheritance and Dynamic Work Class Binding", Proceedings of the Workshop Software Architectures for Business Process Management at the 11th Conference on Advanced Information Systems Engineering (CAiSE*99), Heidelberg, Germany, 1999

[9] G. Kappel, P. Lang, S. Rausch-Schott, W. Retschitzegger; "Work Management Based on Objects, Rules, and Roles", Bulletin of the Technical Committee on Data Engineering, Vol. 18, No. 1, March 1995

[10] M.P. Papazoglou, A. Delis, A. Bouguettaya, M. Haghjoo; "Class Library Support for Work Environments and Applications", IEEE Transactions on Computers, Vol. 46, No.6, June 1997

APPENDIX A - SERVICECOM SCREEN SHOTS

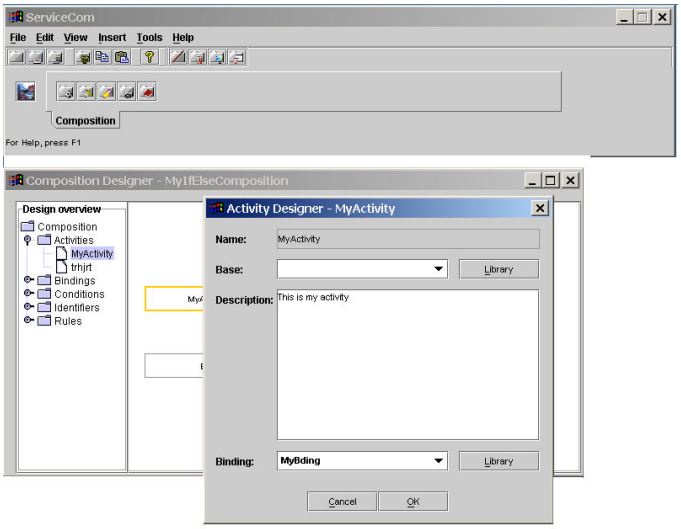

Fig. 6: The main window, the composition designer and the activity designer

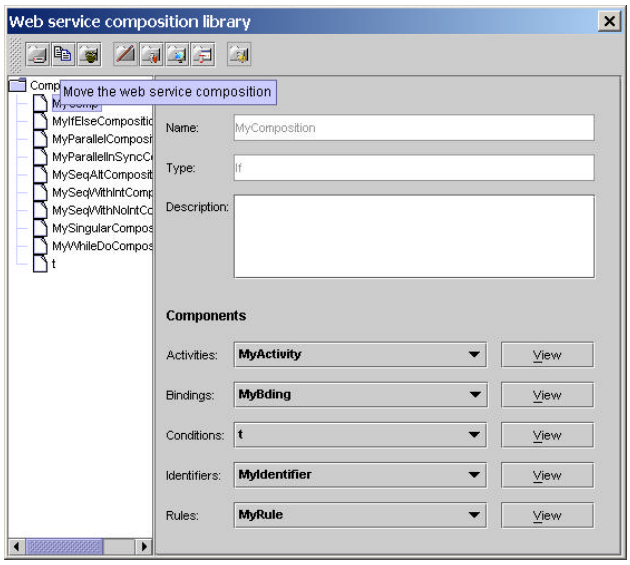

Fig. 8: The web service composition library

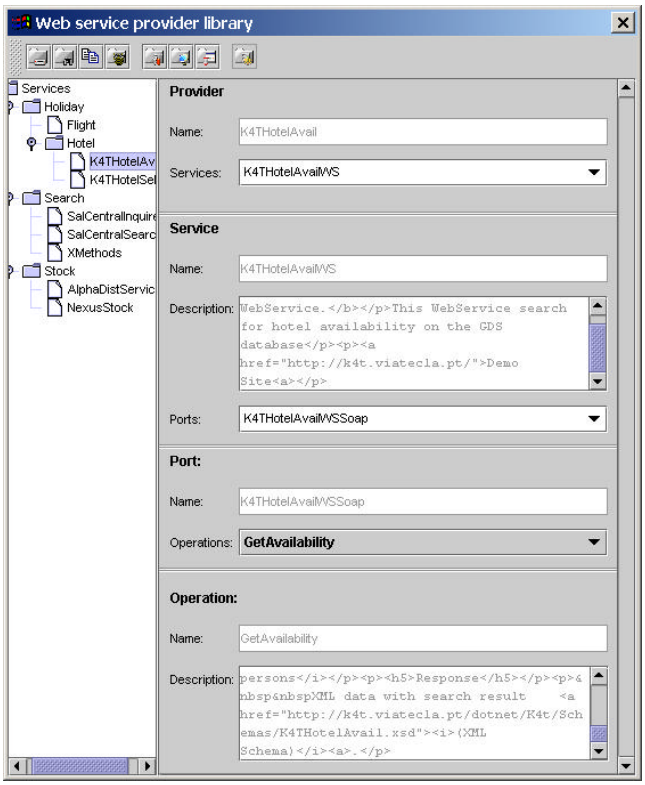

Fig. 7: The web service provider library